VMSP: Efficient Vertical Mining of Maximal Sequential Patterns

Philippe Fournier-Viger[1], Cheng-Wei Wu[2], Antonio Gomariz[3],
and Vincent S. Tseng[2]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Computer Science and Information Engineering, National Cheng Kung
University, Taiwan
[3] Dept. of Information and Communication Engineering, University of Murcia, Spain
`philippe.fournier-viger@umoncton.ca, silvemoonfox@gmail.com,`
`agomariz@um.es, tseng@mail.ncku.edu.tw`

Abstract. *Sequential pattern mining* is a popular data mining task with
wide applications. However, it may present too many sequential patterns
to users, which makes it difficult for users to comprehend the results.
As a solution, it was proposed to mine *maximal sequential patterns*, a
compact representation of the set of sequential patterns, which is of-
ten several orders of magnitude smaller than the set of all sequential
patterns. However, the task of mining maximal patterns remains com-
putationally expensive. To address this problem, we introduce a vertical
mining algorithm named *VMSP* (*Vertical mining of Maximal Sequential
Patterns*). It is to our knowledge the first vertical mining algorithm for
mining maximal sequential patterns. An experimental study on five real
datasets shows that VMSP is up to two orders of magnitude faster than
the current state-of-the-art algorithm.

Keywords: vertical mining, maximal sequential pattern mining, candi-
date pruning.

1 Introduction

Discovering interesting patterns in sequential data is a challenging task. Multiple
studies have been proposed for mining interesting patterns in sequence databases
[11,4]. *Sequential pattern mining* is probably the most popular research topic
among them. A subsequence is called *sequential pattern* or *frequent sequence*
if it frequently appears in a sequence database, and its frequency is no less
than a user-specified *minimum support threshold minsup* [1]. Sequential pattern
mining plays an important role in data mining and is essential to a wide range
of applications such as the analysis of web click-streams, program executions,
medical data, biological data and e-learning data [11].

Several algorithms have been proposed for sequential pattern mining such as
PrefixSpan [12], *SPAM* [2] and *SPADE* [14]. However, a critical drawback of
these algorithms is that they may present too many sequential patterns to users.

A very large number of sequential patterns makes it difficult for users to analyze results to gain insightful knowledge. It may also cause the algorithms to become inefficient in terms of time and memory because the more sequential patterns the algorithms produce, the more resources they consume. The problem becomes worse when the database contains long sequential patterns. For example, consider a sequence database containing a sequential pattern having 20 distinct items. A sequential pattern mining algorithm will present the sequential pattern as well as its $2^{20} - 1$ subsequences to the user. This will most likely make the algorithm fail to terminate in reasonable time and run out of memory. For example, the well-known PrefixSpan [12] algorithm would have to perform 2^{20} database projection operations to produce the results.

To reduce the computational cost of the mining task and present fewer but more representative patterns to users, many studies focus on developing concise representations of sequential patterns. A popular representation that has been proposed is *closed sequential patterns* [13,6]. A closed sequential pattern is a sequential pattern that is not strictly included in another pattern having the same frequency. Several approaches have been proposed for mining closed sequential patterns in sequence databases such as *BIDE* [13] and *ClaSP* [6]. Although these algorithms mines a compact set of sequential patterns, the set of closed patterns is still too large for dense databases or database containing long sequences.

To address this problem, it was proposed to mine *maximal sequential patterns* [3,5,8,9,10,7]. A maximal sequential pattern is a closed pattern that is not strictly included in another closed pattern. The set of maximal sequential patterns is thus generally a very small subset of the set of (closed) sequential patterns. Besides, the set of maximal sequential patterns is representative since it can be used to recover all sequential patterns, and the exact frequency of these latter can also be recovered with a single database pass.

Maximal sequential pattern mining is important and has been adopted in numerous applications. For example, it is used to find the frequent longest common subsequences in texts, analysing DNA sequences, data compression and web log mining [5]. Although maximal sequential pattern mining is desirable and useful in many applications, it remains a computationally expensive data mining task and few algorithms have been proposed for this task. *MSPX* [9] is an approximate algorithm and therefore it provides an incomplete set of maximal patterns to users. *DIMASP* [5] is designed for the special case where sequences are strings (no more than an item can appear at the same time) and where no pair of contiguous items appears more than once in each sequence. *AprioriAdjust* [10] is an apriori-like algorithm, which may suffer from the drawbacks of the candidate generation-and-test paradigm. In other words, it may generate a large number of candidate patterns that do not appear in the input database and require to scan the original database several times. The *MFSPAN* [7] algorithm needs to maintain a large amount of intermediate candidates in main memory during the mining process. The most recent algorithm is *MaxSP* [3], which relies on a pattern-growth approach to avoid the problem of candidate generation

from previous algorithms. However, it has to repeatedly perform costly database projection operations [3].

Given the limitations of previous work, we explore a novel approach, which is to mine maximal sequential pattern by using a depth-first exploration of the search space using a vertical representation. We propose a novel algorithm for maximal sequential pattern mining that we name *VMSP* (*Vertical mining of Maximal Sequential Patterns*). The algorithm incorporates three efficient strategies named EFN (Efficient Filtering of Non-maximal patterns), FME (Forward-Maximal Extension checking) and CPC (Candidate Pruning by Co-occurrence map) to effectively identify maximal patterns and prune the search space. VMSP is developed for the general case of a sequence database rather than strings and it can capture the complete set of maximal sequential patterns with a single database scan. We performed an experimental study with five real-life datasets to compare the performance of VMSP with MaxSP [3], the state-of-the-art algorithm for maximal sequential pattern mining. Results show that VMSP is up to two orders of magnitude faster than MaxSP, and perform well on dense datasets.

The rest of the paper is organized as follows. Section 2 formally defines the problem of maximal sequential pattern mining and its relationship to sequential pattern mining. Section 3 describes the VMSP algorithm. Section 4 presents the experimental study. Finally, Section 5 presents the conclusion and future works.

2   Problem Definition

Definition 1 (sequence database).   Let $I = \{i_1, i_2, ..., i_l\}$ be a set of items (symbols). An *itemset* $I_x = \{i_1, i_2, ..., i_m\} \subseteq I$ is an unordered set of distinct items. The *lexicographical order* $\succ_{lex}$ is defined as any total order on I. Without loss of generality, we assume that all itemsets are ordered according to $\succ_{lex}$. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ ($1 \leq k \leq n$). A *sequence database SDB* is a list of sequences $SDB = \langle s_1, s_2, ..., s_p \rangle$ having sequence identifiers (SIDs) $1, 2...p$. **Example.** A sequence database is shown in Fig. 1 (left). It contains four sequences having the SIDs 1, 2, 3 and 4. Each single letter represents an item. Items between curly brackets represent an itemset. The first sequence $\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$ contains five itemsets. It indicates that items a and b occurred at the same time, were followed by c, then f and g at the same time, followed by g and lastly e.

Definition 2 (sequence containment).   A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is said to be *contained in* a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$ iff there exist integers $1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $A_1 \subseteq B_{i1}, A_2 \subseteq B_{i2}, ..., A_n \subseteq B_{in}$ (denoted as $s_a \sqsubseteq s_b$). **Example.** Sequence 4 in Fig. 1 (left) is contained in Sequence 1.

Definition 3 (prefix).   A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is a *prefix* of a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$, $\forall n < m$, iff $A_1 = B_1, A_2 = B_2, ..., A_{n-1} = B_{n-1}$ and the first $|A_n|$ items of B_n according to $\succ_{lex}$ are equal to A_n.

SID	Sequences
1	$\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$
2	$\langle \{a, d\}, \{c\}, \{b\}, \{a, b, e, f\} \rangle$
3	$\langle \{a\}, \{b\}, \{f, g\}, \{e\} \rangle$
4	$\langle \{b\}, \{f, g\} \rangle$

Pattern	Sup.		Pattern	Sup.	
$\langle \{a\} \rangle$	3	C	$\langle \{b\}, \{g\}, \{e\} \rangle$	2	CM
$\langle \{a\}, \{g\} \rangle$	2		$\langle \{b\}, \{f\} \rangle$	4	C
$\langle \{a\}, \{g\}, \{e\} \rangle$	2	CM	$\langle \{b\}, \{f, g\} \rangle$	2	CM
$\langle \{a\}, \{f\} \rangle$	3	C	$\langle \{b\}, \{f\}, \{e\} \rangle$	2	CM
$\langle \{a\}, \{f\}, \{e\} \rangle$	2	CM	$\langle \{b\}, \{e\} \rangle$	3	C
$\langle \{a\}, \{c\} \rangle$	2		$\langle \{c\} \rangle$	2	
$\langle \{a\}, \{c\}, \{f\} \rangle$	2	CM	$\langle \{c\}, \{f\} \rangle$	2	
$\langle \{a\}, \{c\}, \{e\} \rangle$	2	CM	$\langle \{c\}, \{e\} \rangle$	2	
$\langle \{a\}, \{b\} \rangle$	2		$\langle \{e\} \rangle$	3	
$\langle \{a\}, \{b\}, \{f\} \rangle$	2	CM	$\langle \{f\} \rangle$	4	
$\langle \{a\}, \{b\}, \{e\} \rangle$	2	CM	$\langle \{f, g\} \rangle$	2	
$\langle \{a\}, \{e\} \rangle$	3	C	$\langle \{f\}, \{e\} \rangle$	2	
$\langle \{a, b\} \rangle$	2	CM	$\langle \{g\} \rangle$	3	
$\langle \{b\} \rangle$	4		$\langle \{g\}, \{e\} \rangle$	2	
$\langle \{b\}, \{g\} \rangle$	3	C			

C = Closed   M = Maximal

Fig. 1. A sequence database (left) and (all/closed/maximal) sequential patterns found (right)

Definition 4 (extensions). A sequence s_b is said to be an *s-extension* of a sequence $s_a = \langle I_1, I_2, ...I_h \rangle$ with an item x, iff $s_b = \langle I_1, I_2, ...I_h, \{x\} \rangle$, i.e. s_a is a prefix of s_b and the item x appears in an itemset later than all the itemsets of s_a. In the same way, the sequence s_c is said to be an *i-extension* of s_a with an item x, iff $s_c = \langle I_1, I_2, ...I_h \cup \{x\} \rangle$, i.e. s_a is a prefix of s_c and the item x occurs in the last itemset of s_a, and the item x is the last one in I_h, according to $\succ_{lex}$.

Definition 5 (support). The *support* of a sequence s_a in a sequence database SDB is defined as the number of sequences $s \in SDB$ such that $s_a \sqsubseteq s$ and is denoted by $sup_{SDB}(s_a)$.

Definition 6 (sequential pattern mining). Let $minsup$ be a threshold set by the user and SDB be a sequence database. A sequence s is a *sequential pattern* and is deemed *frequent* iff $sup_{SDB}(s) \geq minsup$. The *problem of mining sequential patterns* is to discover all sequential patterns [1]. **Example.** Fig. 1 (right) shows the 29 sequential patterns found in the database of Fig. 1 (left) for $minsup = 2$, and their support. For instance, the patterns $\langle \{a\} \rangle$ and $\langle \{a\}, \{g\} \rangle$ are frequent and have respectively a support of 3 and 2 sequences.

Definition 7 (closed/maximal sequential pattern mining). A sequential pattern s_a is said to be *closed* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, $s_a \sqsubseteq s_b$, and their supports are equal. A sequential pattern s_a is said to be *maximal* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, $s_a \sqsubseteq s_b$. The *problem of mining closed (maximal) sequential patterns* is to discover the set of closed (maximal) sequential patterns. **Example.** Consider the database of Fig. 1 and minsup = 2. There are 29 sequential patterns (shown in the right side of Fig. 1), such that 15 are closed (identified by the letter C) and only 10 are maximal (identified by the letter M).

Property 1. (Recovering sequential patterns). The set of maximal sequential patterns allow recovering all sequential patterns.

Proof. By definition, a maximal sequential pattern has no proper super-sequence that is a frequent sequential pattern. Thus, if a pattern is frequent, it is either a proper subsequence of a maximal pattern or a maximal pattern. Figure 2 presents a simple algorithm for recovering all sequential patterns from the set of maximal sequential patterns. It generates all the subsequences of all the maximal patterns. Furthermore, it performs a check to detect if a sequential pattern has already been output (line 3) because a sequential pattern may be a subsequence of more than one maximal pattern. After sequential patterns have been recovered, an additional database scan can be performed to calculate their exact support, if required.

RECOVERY (*a set of maximal patterns M*)
1.   **FOR** each sequential pattern $j \in M$,
2.   **FOR** each subsequence k of j,
3.   **IF** k has not been output
4.   **THEN** output k.

Fig. 2. Algorithm to recover all sequential patterns from maximal patterns

Definition 8 (horizontal database format). A *sequence database in horizontal format* is a database where each entry is a sequence. **Example.** Figure 1 (left) shows an horizontal sequence database.

Definition 9 (vertical database format). A *sequence database in vertical format* is a database where each entry represents an item and indicates the list of sequences where the item appears and the position(s) where it appears [2]. **Example.** Fig. 3 shows the vertical representation of the database of Fig. 1 (left).

a		b		c		d	
SID	Itemsets	SID	Itemsets	SID	Itemsets	SID	Itemsets
1	1	1	1	1	2	1	
2	1,4	2	3,4	2	2	2	1
3	1	3	2	3		3	
4		4	1	4		4	

e		f		g	
SID	Itemsets	SID	Itemsets	SID	Itemsets
1	5	1	3	1	3,4
2	4	2	4	2	
3	4	3	3	3	
4		4	2	4	2

Fig. 3. The vertical representation of the database shown in Fig. 1(left)

Vertical mining algorithms associate a structure named *IdList* [14,2] to each pattern. IdLists allow calculating the support of a pattern quickly by making

join operations with IdLists of smaller patterns. To discover sequential patterns, vertical mining algorithms perform a single database scan to create IdLists of patterns containing single items. Then, larger patterns are obtained by performing the join operation of IdLists of smaller patterns (cf. [14] for details). Several works proposed alternative representations for IdLists to save time in join operations, being the bitset representation the most efficient one [2].

3   The VMSP Algorithm

We present VMSP, our novel algorithm for maximal sequential pattern mining. It adopts the IdList structure [2,6,14]. We first describe the general search procedure used by VMSP to explore the search space of sequential patterns. Then, we describe how it is adapted to discover maximal patterns efficiently.

3.1   The Search Procedure

The pseudocode of the search procedure is shown in Fig. 4. The procedure takes as input a sequence database SDB and the $minsup$ threshold. The procedure first scans the input database SDB once to construct the vertical representation of the database $V(SDB)$ and the set of frequent items F_1. For each frequent item $s \in F_1$, the procedure calls the SEARCH procedure with $\langle s \rangle$, F_1, $\{e \in F_1 | e \succ_{lex} s\}$, and $minsup$.

The SEARCH procedure outputs the pattern $\langle \{s\} \rangle$ and recursively explores candidate patterns starting with the prefix $\langle \{s\} \rangle$. The SEARCH procedure takes as parameters a sequential pattern pat and two sets of items to be appended to pat to generate candidates. The first set S_n represents items to be appended to pat by s-extension. The second set S_i represents items to be appended to pat by i-extension. For each candidate pat' generated by an extension, the procedure calculate the support to determine if it is frequent. This is done by the IdList join operation (see [2,14] for details) and counting the number of sequences where the pattern appears. If the pattern pat' is frequent, it is then used in a recursive call to SEARCH to generate patterns starting with the prefix pat'.

It can be easily seen that the above procedure is correct and complete to explore the search space of sequential patterns since it starts with frequent patterns containing single items and then extend them one item at a time while only pruning infrequent extensions of patterns using the anti-monotonicity property (any infrequent sequential pattern cannot be extended to form a frequent pattern)[1].

3.2   Discovering Maximal Patterns

We now describe how the search procedure is adapted to discover only maximal patterns. This is done by integrating three strategies to efficiently filter non-maximal patterns and prune the search space. The result is the VMSP algorithm, which outputs the set of maximal patterns.

PATTERN-ENUMERATION(*SDB, minsup*)
1. Scan *SDB* to create *V*(*SDB*) and identify S_{init}, the list of frequent items.
2. **FOR** each item s ∈ S_{init},
3.   **SEARCH**($\langle s \rangle$), S_{init}, the set of items from S_{init} that are lexically larger than *s, minsup*).

SEARCH(*pat, S_n, I_n, minsup*)
1. Output pattern *pat.*
2. $S_{temp} := I_{temp} := \emptyset$
3. **FOR** each item $j \in S_n$,
4.   **IF** the s-extension of *pat* is frequent **THEN** $S_{temp} := S_{temp} \cup \{i\}$.
5. **FOR** each item $j \in S_{temp}$,
6.   **SEARCH**(the s-extension of *pat* with *j, S_{temp}*, elements in S_{temp} greater than *j, minsup*).
7. **FOR** each item $j \in I_n$,
8.   **IF** the i-extension of *pat* is frequent **THEN** $I_{temp} := I_{temp} \cup \{i\}$.
9. **FOR** each item $j \in I_{temp}$,
10.   **SEARCH**(i-extension of *pat* with *j, S_{temp}*, all elements in I_{temp} greater than *j, minsup*).

Fig. 4. The search procedure

Strategy 1. Efficient Filtering of Non-maximal patterns (EFN). The first strategy identifies maximal patterns among patterns generated by the search procedure. This is performed using a novel structure named Z that stores the set of maximal patterns found until now. The structure Z is initially empty. Then, during the search for patterns, every time that a pattern s_a, is generated by the search procedure, two operations are performed to update Z.

- *Super-pattern checking.* During this operation, s_a is compared with each pattern $s_b \in Z$ to determine if there exists a pattern s_b such that $s_a \sqsubseteq s_b$. If yes, then s_a is not maximal (by Definition 7) and thus, s_a is not inserted into Z. Otherwise, s_a is maximal with respect to all patterns found until now and it is thus inserted into Z.
- *Sub-pattern checking.* If s_a is determined to be maximal according to super-pattern checking, we need to perform this second operation. The pattern s_a is compared with each pattern $s_b \in Z$. If there exists a pattern $s_b \sqsubseteq s_a$, then s_b is not maximal (by Definition 7) and s_b is removed from Z.

By using the above strategy, it is obvious that when the search procedure terminates, Z contains the set of maximal sequential patterns. However, to make this strategy efficient, we need to reduce the number of pattern comparisons and containment checks ($\sqsubseteq$). We propose three optimizations.

1. *Size check optimization.* Let n be the number of items in the largest pattern found until now. The structure Z is implemented as a list of heaps $Z = \{Z_1, Z_2, ... Z_n\}$, where Z_x contains all maximal patterns found until now having x items ($1 \leq x \leq n$). To perform sub-pattern checking (super-pattern checking) for a pattern s containing w items, an optimization is to only compare s with patterns in $Z_1, Z_2 ... Z_{w-1}$ (in $Z_{w+1}, Z_{w+2} ... Z_n$) because a pattern can only contain (be contained) in smaller (larger) patterns.

2. *Sum of items optimization.* In our implementation, each item is represented by an integer. For each pattern s, the *sum of the items* appearing in the pattern is computed, denoted as $sum(s)$. In each heap, patterns are ordered by decreasing sum of items. This allows the following optimization. Consider super-pattern checking for a pattern s_a and a heap Z_x. If $sum(s_a) < sum(s_b)$ for a pattern s_b in Z_x, then we don't need to check $s_a \sqsubseteq s_b$ for s_b and all patterns following s_b in Z_x. A similar optimization is done for sub-pattern checking. Consider sub-pattern checking for a pattern s_a and a heap Z_x. If $sum(s_b) < sum(s_a)$ for a pattern s_b in Z_x, then we don't need to check $s_b \sqsubseteq s_a$ for s_b and all patterns following s_b in Z_x, given that Z_x is traversed in reverse order.

3. *Support check optimization.* This optimization uses the support to avoid containment checks ($\sqsubseteq$). If the support of a pattern s_a is less than the support of another pattern s_b (greater), then we skip checking $s_a \sqsubseteq s_b$ ($s_b \sqsubseteq s_a$).

Strategy 2. Forward-Maximal Extension checking (FME). The second strategy aims at avoiding super-pattern checks. The search procedure discovers patterns by growing a pattern by appending one item at a time by s-extension or i-extension. Consider a pattern x that is generated. An optimization is to not perform super-pattern checking if the recursive call to the SEARCH procedure generate a frequent pattern (because this pattern would have x has prefix, thus indicating that x is not maximal).

Strategy 3. Candidate Pruning with Co-occurrence map (CPC). The last strategy aims at pruning the search space of patterns by exploiting item co-occurrence information. We introduce a structure named *Co-occurrence MAP* (CMAP) defined as follows: an item k is said to *succeed by i-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j, k \in I_x$ for an integer x such that $1 \leq x \leq n$ and $k \succ_{lex} j$. In the same way, an item k is said to *succeed by s-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j \in I_v$ and $k \in I_w$ for some integers v and w such that $1 \leq v < w \leq n$. A CMAP is a structure mapping each item $k \in I$ to a set of items succeeding it.

We define two CMAPs named $CMAP_i$ and $CMAP_s$. $CMAP_i$ maps each item k to the set $cm_i(k)$ of all items $j \in I$ succeeding k by i-extension in no less than *minsup* sequences of SDB. $CMAP_s$ maps each item k to the set $cm_s(k)$ of all items $j \in I$ succeedings k by s-extension in no less than *minsup* sequences of SDB. For example, the $CMAP_i$ and $CMAP_s$ structures built for the sequence database of Fig. 1(left) are shown in Table 1. Both tables have been created considering a *minsup* of two sequences. For instance, for the item f, we can see that it is associated with an item, $cm_i(f) = \{g\}$, in $CMAP_i$, whereas it is associated with two items, $cm_s(f) = \{e, g\}$, in $CMAP_s$. This indicates that both items e and g succeed to f by s-extension and only item g does the same for i-extension, being all of them in no less than *minsup* sequences.

VMSP uses CMAPs to prune the search space as follows:

1. *s-extension(s) pruning.* Let a sequential pattern *pat* being considered for *s*-extension with an item $x \in S_n$ by the SEARCH procedure (line 3). If the last item a in *pat* does not have an item $x \in cm_s(a)$, then clearly the pattern resulting from the extension of *pat* with x will be infrequent and thus the join operation of x with *pat* to count the support of the resulting pattern does not need to be performed. Furthermore, the item x is not considered for generating any pattern by *s*-extension having *pat* as prefix, by not adding x to the variable S_{temp} that is passed to the recursive call to the SEARCH procedure. Moreover, note that we only have to check the extension of *pat* with x for the last item in *pat*, since other items have already been checked for extension in previous steps.
2. *i-extension(s) pruning.* Let a sequential pattern *pat* being considered for *i*-extension with an item $x \in I_n$ by the SEARCH procedure. If the last item a in *pat* does not have an item $x \in cm_i$, then clearly the pattern resulting from the extension of *pat* with x will be infrequent and thus the join operation of x with *pat* to count the support of the resulting pattern does not need to be performed. Furthermore, the item x is not considered for generating any pattern by *i*-extension(s) of *pat* by not adding x to the variable I_{temp} that is passed to the recursive call to the SEARCH procedure. As before, we only have to check the extension of *pat* with x for the last item in *pat*, since others have already been checked for extension in previous steps.

CMAPs are easily maintained and are built with a single database scan. With regards to their implementation, we define each one as a hash table of hash sets, where an hashset corresponding to an item k only contains the items that succeed to k in at least *minsup* sequences.

Table 1. $CMAP_i$ and $CMAP_s$ for the database of Fig. 1 and $minsup = 2$

$CMAP_i$		$CMAP_s$	
item	is succeeded by (i-extension)	item	is succeeded by (s-extension)
a	$\{b\}$	a	$\{b, c, e, f\}$
b	$\emptyset$	b	$\{e, f, g\}$
c	$\emptyset$	c	$\{e, f\}$
e	$\emptyset$	e	$\emptyset$
f	$\{g\}$	f	$\{e, g\}$
g	$\emptyset$	g	$\emptyset$

Lastly, since the VMSP algorithm is a vertical mining algorithm, it relies on IDLists. We implement IDLists as bitsets as it is done in several state-of-art algorithms [2,6]. Bitsets speed up the join operations. Algorithms using this representation were demonstrated to be much faster than vertical mining algorithms which do not use them.

4   Experimental Evaluation

We performed several experiments to assess the performance of the proposed algorithm. Experiments were performed on a computer with a third generation Core i5 64 bit processor running Windows 7 and 5 GB of free RAM. We compared the performance of VMSP with MaxSP, the current state-of-the-art algorithm for maximal sequential pattern mining. All algorithms were implemented in Java. All memory measurements were done using the Java API. Experiments were carried on five real-life datasets having varied characteristics and representing three different types of data (web click stream, text from a book and protein sequences). Those datasets are *Leviathan*, *Snake*, *FIFA*, *BMS* and *Kosarak10k*. Table 2 summarizes their characteristics. The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/hDtdt`.

Table 2. Dataset characteristics

dataset	sequence count	item count	avg. seq. length (items)	type of data
Leviathan	5834	9025	33.81 (std= 18.6)	book
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream
BMS	59601	497	2.51 (std = 4.85)	web click stream
Kosarak10k	10000	10094	8.14 (std = 22)	web click stream

Experiment 1. Influence of the *minsup* parameter. The first experiment consisted of running all the algorithms on each dataset while decreasing the *minsup* threshold until an algorithm became too long to execute, ran out of memory or a clear winner was observed. For each dataset, we recorded the execution time and memory usage.

In terms of execution time, results (cf. Fig. 5) show that VMSP outperforms MaxSP by a wide margin on all datasets. Moreover, VMSP performs very well on dense datasets (about 100 times faster than MaxSP on Snake and FIFA).

In terms of memory consumption the maximum memory usage of VMSP (MaxSP) on BMS, Snake, Kosarak, Leviathan and FIFA was respectively 840 MB (403 MB), 45 MB (340 MB), 1600 MB (393 MB), 911 MB (1150 MB) and 611 MB (970 MB). Overall, VMSP has the lowest memory consumption for three out of five datasets.

Experiment 2. Influence of the strategies. We next evaluated the benefit of using strategies in VMSP. We compared VMSP with a version of VMSP without strategy CPC (VMSP_W3), a version without strategies FME and CPC (VMSP_W2W3), and a version of VMSP without FME, CPC and strategies in EFN (VMSP_W1W2W3). Results for the BMS and FIFA datasets are shown in Fig. 6. Results for other datasets are similar and are not shown due to space limitation. As a whole, strategies improved execution time by up to to 8 times, CPC being the most effective strategy.
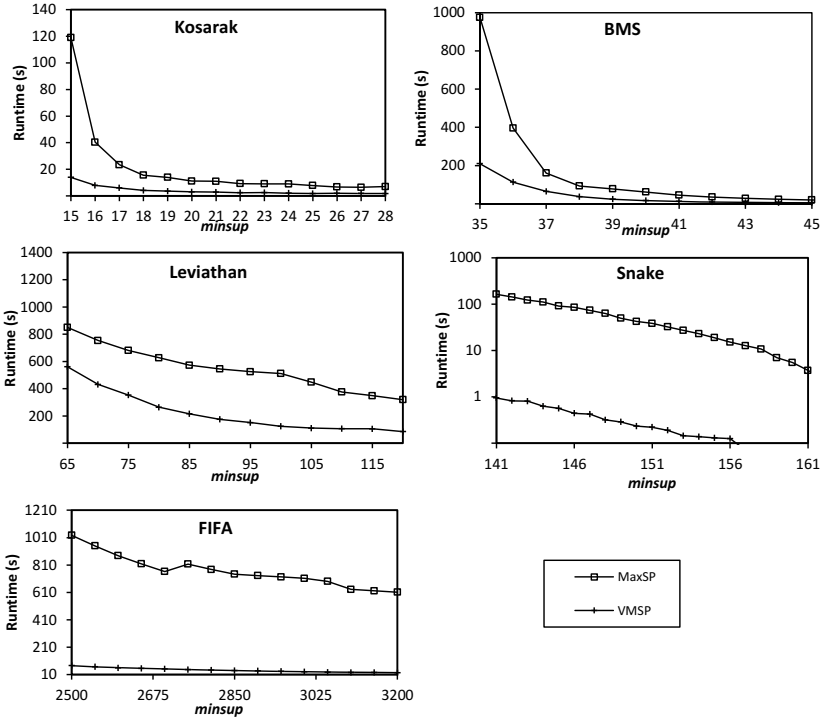
Fig. 5. Execution times

We also measured the memory used by the CPC strategy to build the CMAPs data structure. We found that the required amount memory is very small. For the BMS, Kosarak, Leviathan, Snake and FIFA datasets, the memory footprint of CMAPs was respectively 0.5 MB, 33.1 MB, 15 MB, 64 KB and 0.4 MB.
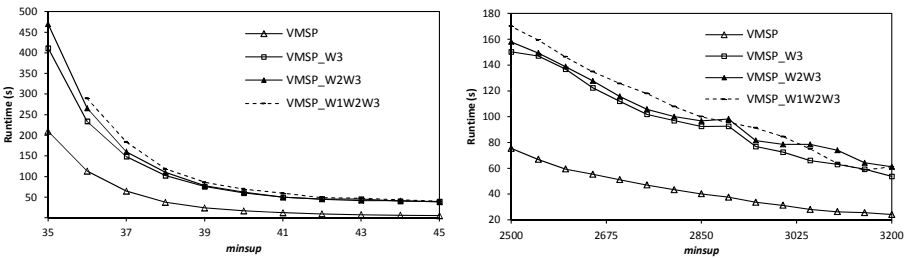

Fig. 6. Influence of optimizations for BMS (left) and FIFA (right)

5   Conclusion

In this paper, we presented a new maximal sequential pattern mining algorithm named VMSP (Vertical Maximal Sequential Pattern miner). It is to our knowledge the first vertical algorithm for this task. Furthermore, it includes three novel strategies for efficiently identifying maximal patterns and pruning the search space (EFN, FME and CPC). An experimental study on five real datasets shows that VMSP is up to two orders of magnitude faster than MaxSP, the state-of-art algorithm for maximal sequential pattern mining, and that VMSP performs well on dense datasets. The source code of VMSP and MaxSP can be downloaded from `http://goo.gl/hDtdt`.

References

1. Agrawal, R., Ramakrishnan, S.: Mining sequential patterns. In: Proc. 11th Intern. Conf. Data Engineering, pp. 3–14. IEEE (1995)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proc. 8th ACM Intern. Conf. Knowl. Discov. Data Mining, pp. 429–435. ACM (2002)
3. Fournier-Viger, P., Wu, C.-W., Tseng, V.S.: Mining Maximal Sequential Patterns without Candidate Maintenance. In: Motoda, H., Wu, Z., Cao, L., Zaiane, O., Yao, M., Wang, W. (eds.) ADMA 2013, Part I. LNCS (LNAI), vol. 8346, pp. 169–180. Springer, Heidelberg (2013)
4. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast Vertical Sequential Pattern Mining Using Co-occurrence Information. In: Proc. 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining. LNCS (LNAI). Springer (2014)
5. García-Hernández, R.A., Martínez-Trinidad, J.F., Carrasco-Ochoa, J.A.: A new algorithm for fast discovery of maximal sequential patterns in a document collection. In: Gelbukh, A. (ed.) CICLing 2006. LNCS, vol. 3878, pp. 514–523. Springer, Heidelberg (2006)
6. Gomariz, A., Campos, M., Marin, R., Goethals, B.: ClaSP: An Efficient Algorithm for Mining Frequent Closed Sequences. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds.) PAKDD 2013, Part I. LNCS, vol. 7818, pp. 50–61. Springer, Heidelberg (2013)
7. Guan, E.-Z., Chang, X.-Y., Wang, Z., Zhou, C.-G.: Mining Maximal Sequential Patterns. In: Proc. 2nd Intern. Conf. Neural Networks and Brain, pp. 525–528 (2005)
8. Lin, N.P., Hao, W.-H., Chen, H.-J., Chueh, H.-E., Chang, C.-I.: Fast Mining Maximal Sequential Patterns. In: Proc. of the 7th Intern. Conf. on Simulation, Modeling and Optimization, Beijing, China, September 15-17, pp. 405–408 (2007)
9. Luo, C., Chung, S.: Efficient mining of maximal sequential patterns using multiple samples. In: Proc. 5th SIAM Intern. Conf. Data Mining, Newport Beach, CA (2005)
10. Lu, S., Li, C.: Apriori Adjust: An Efficient Algorithm for Discovering the Maximum Sequential Patterns. In: Proc. Intern. Workshop Knowl. Grid and Grid Intell. (2004)
11. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms. ACM Computing Surveys 43(1), 1–41 (2010)
12. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans. Known. Data Engin. 16(11), 1424–1440 (2004)
13. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. IEEE Trans. on Knowledge Data Engineering 19(8), 1042–1056 (2007)
14. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42(1), 31–60 (2001)